Component Reconfiguration in the Presence of Conflicts*

Roberto Di Cosmo[1], Jacopo Mauro[2], Stefano Zacchiroli[1], and Gianluigi Zavattaro[2]

[1] Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France
roberto@dicosmo.org, zack@pps.univ-paris-diderot.fr
[2] Focus Team, Univ of Bologna/INRIA, Italy, Mura A. Zamboni, 7, Bologna
{jmauro,zavattar}@cs.unibo.it

Abstract. Components are traditionally modeled as black-boxes equipped with interfaces that indicate provided/required ports and, often, also conflicts with other components that cannot coexist with them. In modern tools for automatic system management, components become *grey*-boxes that show relevant internal states and the possible actions that can be acted on the components to change such state during the deployment and reconfiguration phases. However, state-of-the-art tools in this field do not support a systematic management of conflicts. In this paper we investigate the impact of conflicts by precisely characterizing the increment of complexity on the reconfiguration problem.

1 Introduction

Modern software systems are more and more based on interconnected software components (e.g. packages or services) deployed on clusters of heterogeneous machines that can be created, connected and reconfigured on-the-fly. Traditional component models represent components as black-boxes with interfaces indicating their *provide* and *require* ports. In many cases also *conflicts* are considered in order to deal with frequent situations in which components cannot be co-installed.

In software systems where components are frequently reconfigured (e.g. "cloud" based applications that elastically reacts to client demands) more expressive component models are considered: a component becomes a grey-box showing relevant internal states and the actions that can be acted on the component to change state during deployment and reconfiguration. For instance, in the popular system configuration tool Puppet [10] or the novel deployment management system Engage [8], components can be in the *absent*, *present*, *running* or *stopped* states, and the actions *install*, *uninstall*, *start*, *stop* and *restart* can be executed upon them. Rather expressive dependencies among components can be declared. The aim of these tools is to allow the system administrator to declaratively express the desired component configuration and automatically execute a correct sequence of low-level actions that bring the current configuration to a new one satisfying the administrator requests respecting dependencies. We call *reconfigurability* the problem of checking the existence of such sequence of low-level actions.

* Work partially supported by Aeolus project, ANR-2010-SEGI-013-01, and performed at IRILL, center for Free Software Research and Innovation in Paris, France, www.irill.org

F.V. Fomin et al. (Eds.): ICALP 2013, Part II, LNCS 7966, pp. 187–198, 2013.

Despite the importance of conflicts in many component models, see e.g. package-based software distributions used for Free and Open Source Software (FOSS) [5], the Eclipse plugin model [3], or the OSGi component framework [12], state-of-the-arts management systems like the above do not take conflicts into account. This is likely ascribable to the increased complexity of the reconfigurability problem in the presence of conflicts. In this paper we precisely characterize this increment of complexity.

In a related paper [6] we have proposed the Aeolus component model that, despite its simplicity, is expressive enough to capture the main features of tools like Puppet and Engage. We have proved that the reconfigurability problem is Polynomial-Time for Aeolus$^-$, the fragment without numerical constraints. In this paper we consider Aeolus core, the extension of this fragment with conflicts, and we prove that even if the reconfigurability problem remains decidable, it turns out to be Exponential-Space hard. We consider this result a fundamental step towards the realization of tools that manage conflicts systematically. In fact, we shed some light on the specific sources of the increment of complexity of the reconfigurability problem.

The technical contribution of the paper and its structure is as follows. In Section 2 we formalize the reconfigurability problem in the presence of conflicts. In Section 3 we prove its decidability by resorting to the theory of Well-Structured Transition Systems [2,7]. We consider this decidability result interesting also from a foundational viewpoint: despite our component model has many commonalities with concurrent models like Petri nets, in our case the addition of conflicts (corresponding to inhibitor arcs in Petri nets) does not make the analysis of reachability problems undecidable. The closed relationship between our model and Petri nets is used in Section 4 where we prove the Exponential-Space hardness of the reconfigurability problem by reduction from the coverability problem in Petri nets. In Section 5 we discuss related work and report concluding remarks. Missing proofs are available in [4].

2 The Aeolus core Model

The Aeolus core model represents relevant internal states of components by means of a *finite state automaton* (see Fig. 1): depending on its state components activate *provide* and *require* functionalities (called *ports*), and get in *conflict* with ports provided by others (in Fig. 1 active ports are black while inactive ones are grey). Each port is identified by an interface name. Bindings can be established between provide and require ports with the same interface. Fig. 1 shows the graphical representation of a typical deployment of the popular WordPress blog. According to the Debian packages metadata, WordPress requires a Web server providing httpd in order to be installed, and an active MySQL database server in order to be in production. The chosen Web server is Apache2 which is broken into various packages (e.g. `apache2`, `apache2-bin`) that shall be simultaneously installed. Notice that Apache2 is not co-installable with other Web servers, such as lighttpd.

We now move to the formal definition of Aeolus core. We assume given a set $\mathscr{I}$ of interface names.

Definition 1 (Component type). *The set Γ of* component types *of the* Aeolus core *model, ranged over by $\mathscr{T}, \mathscr{T}_1, \mathscr{T}_2, \ldots$ contains 4-ples $\langle Q, q_0, T, D \rangle$ where:*

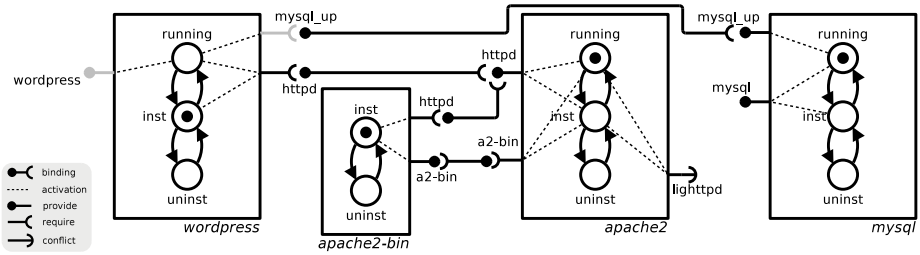

Fig. 1. Typical Wordpress/Apache/MySQL deployment, modeled in Aeolus core

- *Q is a finite set of states containing the initial state q_0;*
- *$T \subseteq Q \times Q$ is the set of* transitions*;*
- *D is a function from Q to a 3-ple $\langle \mathbf{P}, \mathbf{R}, \mathbf{C} \rangle$ of interface names (i.e. $\mathbf{P}, \mathbf{R}, \mathbf{C} \subseteq \mathscr{I}$) indicating the provide, require, and conflict ports that each state activates. We assume that the initial state q_0 has no requirements and conflicts (i.e. $D(q_0) = \langle \mathbf{P}, \emptyset, \emptyset \rangle$).*

We now define configurations that describe systems composed by components and their bindings. Each component has a unique identifier taken from the set $\mathscr{Z}$. A configuration, ranged over by $\mathscr{C}_1, \mathscr{C}_2, \ldots$, is given by a set of component types, a set of components in some state, and a set of bindings.

Definition 2 (Configuration). *A* configuration *$\mathscr{C}$ is a 4-ple $\langle U, Z, S, B \rangle$ where:*

- *$U \subseteq \Gamma$ is the finite* universe *of the available component types;*
- *$Z \subseteq \mathscr{Z}$ is the set of the currently deployed* components*;*
- *S is the component* state description*, i.e. a function that associates to components in Z a pair $\langle \mathscr{T}, q \rangle$ where $\mathscr{T} \in U$ is a component type $\langle Q, q_0, T, D \rangle$, and $q \in Q$ is the current component state;*
- *$B \subseteq \mathscr{I} \times Z \times Z$ is the set of* bindings*, namely 3-ple composed by an interface, the component that requires that interface, and the component that provides it; we assume that the two components are different.*

Configuration are equivalent if they have the same instances up to instance renaming.

Definition 3 (Configuration equivalence). *Two configurations $\langle U, Z, S, B \rangle$ and $\langle U, Z', S', B' \rangle$ are equivalent ($\langle U, Z, S, B \rangle \equiv \langle U, Z', S', B' \rangle$) iff there exists a bijective function ρ from Z to Z' s.t.*

- *$S(z) = S'(\rho(z))$ for every $z \in Z$;*
- *$\langle r, z_1, z_2 \rangle \in B$ iff $\langle r, \rho(z_1), \rho(z_2) \rangle \in B'$.*

Notation. We write $\mathscr{C}[z]$ as a lookup operation that retrieves the pair $\langle \mathscr{T}, q \rangle = S(z)$, where $\mathscr{C} = \langle U, Z, S, B \rangle$. On such a pair we then use the postfix projection operators `.type` and `.state` to retrieve $\mathscr{T}$ and q, respectively. Similarly, given a component type $\langle Q, q_0, T, D \rangle$, we use projections to decompose it: `.states`, `.init`, and `.trans` return the first three elements; $.\mathbf{P}(q)$, $.\mathbf{R}(q)$, and $.\mathbf{C}(q)$ return the three elements of the $D(q)$ tuple. Moreover, we use `.prov` (resp. `.req`) to denote the union of all the provide ports (resp. require ports) of the states in Q. When there is no

ambiguity we take the liberty to apply the component type projections to $\langle \mathscr{T}, q \rangle$ pairs. *Example*: $\mathscr{C}[z].\mathbf{R}(q)$ stands for the require ports of component z in configuration $\mathscr{C}$ when it is in state q.

We can now formalize the notion of configuration correctness.

Definition 4 (Correctness). *Let us consider the configuration* $\mathscr{C} = \langle U, Z, S, B \rangle$.

We write $\mathscr{C} \models_{req} (z, r)$ *to indicate that the require port of component z, with interface r, is bound to an active port providing r, i.e. there exists a component* $z' \in Z \setminus \{z\}$ *such that* $\langle r, z, z' \rangle \in B$, $\mathscr{C}[z'] = \langle \mathscr{T}', q' \rangle$ *and r is in* $\mathscr{T}'.\mathbf{P}(q')$. *Similarly, for conflicts, we write* $\mathscr{C} \models_{cnf} (z, c)$ *to indicate that the conflict port c of component z is satisfied because no other component has an active port providing c, i.e. for every* $z' \in Z \setminus \{z\}$ *with* $\mathscr{C}[z'] = \langle \mathscr{T}', q' \rangle$ *we have that* $c \notin \mathscr{T}'.\mathbf{P}(q')$.

The configuration $\mathscr{C}$ *is* correct *if for every component* $z \in Z$ *with* $S(z) = \langle \mathscr{T}, q \rangle$ *we have that* $\mathscr{C} \models_{req} (z, r)$ *for every* $r \in \mathscr{T}.\mathbf{R}(q)$ *and* $\mathscr{C} \models_{cnf} (z, c)$ *for every* $c \in \mathscr{T}.\mathbf{C}(q)$.

Configurations evolve at the granularity of actions.

Definition 5 (Actions). *The set* $\mathscr{A}$ *contains the following actions:*

- $stateChange(\langle z_1, q_1, q_1' \rangle, \ldots, \langle z_n, q_n, q_n' \rangle)$ *where* $z_i \in \mathscr{Z}$ *and* $\forall i \neq j \,.\, z_i \neq z_j$;
- $bind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathscr{Z}$ *and* $r \in \mathscr{I}$;
- $unbind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathscr{Z}$ *and* $r \in \mathscr{I}$;
- $newRsrc(z : \mathscr{T})$ *where* $z \in \mathscr{Z}$ *and* $\mathscr{T} \in U$ *is the component type of z;*
- $delRsrc(z)$ *where* $z \in \mathscr{Z}$.

Notice that we consider a set of state changes in order to deal with simultaneous installations like the one needed for Apache2 and Apache2-bin in Fig. 1. The execution of actions is formalized as configuration transitions.

Definition 6 (Reconfigurations). *Reconfigurations are denoted by transitions* $\mathscr{C} \xrightarrow{\alpha} \mathscr{C}'$ *meaning that the execution of* $\alpha \in \mathscr{A}$ *on the configuration* $\mathscr{C}$ *produces a new configuration* $\mathscr{C}'$. *The transitions from a configuration* $\mathscr{C} = \langle U, Z, S, B \rangle$ *are defined as follows:*

$$\mathscr{C} \xrightarrow{stateChange(\langle z_1, q_1, q_1' \rangle, \ldots, \langle z_n, q_n, q_n' \rangle)} \langle U, Z, S', B \rangle$$
$$\text{if } \forall i \,.\, \mathscr{C}[z_i].\texttt{state} = q_i$$
$$\text{and } \forall i \,.\, (q_i, q_i') \in \mathscr{C}[z_i].\texttt{trans}$$
$$\text{and } S'(z') = \begin{cases} \langle \mathscr{C}[z_i].\texttt{type}, q_i' \rangle & \text{if } \exists i \,.\, z' = z_i \\ \mathscr{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathscr{C} \xrightarrow{bind(r, z_1, z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2 \rangle \rangle$$
$$\text{if } \langle r, z_1, z_2 \rangle \notin B$$
$$\text{and } r \in \mathscr{C}[z_1].\texttt{req} \cap \mathscr{C}[z_2].\texttt{prov}$$

$$\mathscr{C} \xrightarrow{unbind(r, z_1, z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2 \rangle \rangle \quad \text{if } \langle r, z_1, z_2 \rangle \in B$$

$$\mathscr{C} \xrightarrow{newRsrc(z:\mathscr{T})} \langle U, Z \cup \{z\}, S', B \rangle$$
$$\text{if } z \notin Z,\ \mathscr{T} \in U$$
$$\text{and } S'(z') = \begin{cases} \langle \mathscr{T}, \mathscr{T}.\texttt{init} \rangle & \text{if } z' = z \\ \mathscr{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathscr{C} \xrightarrow{delRsrc(z)} \langle U, Z \setminus \{z\}, S', B' \rangle$$
$$\text{if } S'(z') = \begin{cases} \bot & \text{if } z' = z \\ \mathscr{C}[z'] & \text{otherwise} \end{cases}$$
$$\text{and } B' = \{ \langle r, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\} \}$$

We can now define a *reconfiguration run* as the effect of the execution of a sequence of actions (atomic or multiple state changes).

Definition 7 (Reconfiguration Run). *A* reconfiguration run *is a sequence of reconfigurations* $\mathscr{C}_0 \xrightarrow{\alpha_1} \mathscr{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathscr{C}_m$ *such that* $\mathscr{C}_i$ *is correct, for every* $0 \leq i \leq m$.

As an example, a reconfiguration run to reach the scenario depicted in Fig. 1 starting from a configuration where only `apache2` and `mysql` are running and `apache2-bin` is installed is the one involving in sequence the creation of `wordpress`, the bindings of `wordpress` with `mysql` and `apache2`, and finally the installation of `wordpress`.

We now have all the ingredients to define the *reconfigurability* problem: given a universe of component types and an initial configuration, we want to know whether there exists a reconfiguration run leading to a configuration that includes at least one component of a given type $\mathscr{T}$ in a given state q.

Definition 8 (Reconfigurability Problem). *The* reconfigurability problem *has as input a universe U of component types, an initial configuration* $\mathscr{C}$*, a component type* $\mathscr{T}$*, and a state q. It returns as output* ***true*** *if there exists a reconfiguration run* $\mathscr{C} \xrightarrow{\alpha_1} \mathscr{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathscr{C}_m$ *and* $\mathscr{C}_m[z] = \langle \mathscr{T}, q \rangle$*, for some component* $z \in \mathscr{C}_m$*. Otherwise, it returns* ***false****.*

The restriction to only one component in a given state is not limiting: we can encode any given combination of component types and states by adding dummy provide ports enabled only by the final states of interest, and a target dummy component with requirements on all such provide ports.

3 Reconfigurability is Decidable in Aeolus core

We demonstrate decidability of the reconfigurability problem by resorting to the theory of Well-Structured Transition Systems (WSTS) [2,7].

A reflexive and transitive relation is called *quasi-ordering*. A *well-quasi-ordering* (wqo) is a quasi-ordering $(X, \leq)$ such that, for every infinite sequence $x_1, x_2, x_3, \cdots$, there exist $i < j$ with $x_i \leq x_j$. Given a quasi-order $\leq$ over X, an *upward-closed set* is a subset $I \subseteq X$ such that the following holds: $\forall x, y \in X : (x \in I \wedge x \leq y) \Rightarrow y \in I$. Given $x \in X$, its upward closure is $\uparrow x = \{y \in X \mid x \leq y\}$. This notion can be extended to sets in the obvious way: given a set $Y \subseteq X$ we define its upward closure as $\uparrow Y = \bigcup_{y \in Y} \uparrow y$. A *finite basis* of an upward-closed set I is a finite set B such that $I = \bigcup_{x \in B} \uparrow x$.

Definition 9. *A WSTS is a transition system* $(\mathscr{S}, \rightarrow, \preceq)$ *where* $\preceq$ *is a wqo on* $\mathscr{S}$ *which is* compatible *with* $\rightarrow$*, i.e., for every* $s_1 \preceq s_1'$ *such that* $s_1 \rightarrow s_2$*, there exists* $s_1' \rightarrow^* s_2'$ *such that* $s_2 \preceq s_2'$ *(*$\rightarrow^*$ *is the reflexive and transitive closure of* $\rightarrow$*). Given a state* $s \in \mathscr{S}$*,* $Pred(s)$ *is the set* $\{s' \in \mathscr{S} \mid s' \rightarrow s\}$ *of immediate predecessors of s. Pred is extended to sets in the obvious way:* $Pred(S) = \bigcup_{s \in S} Pred(s)$*. A WSTS has* effective pred-basis *if there exists an algorithm that, given* $s \in \mathscr{S}$*, returns a finite basis of* $\uparrow Pred(\uparrow s)$.

The following proposition is a special case of Proposition 3.5 in [7].

Proposition 1. *Let* $(\mathscr{S}, \rightarrow, \preceq)$ *be a finitely branching WSTS with decidable* $\preceq$ *and effective pred-basis. Let I be any upward-closed subset of* $\mathscr{S}$ *and let* $Pred^*(I)$ *be the set* $\{s' \in S \mid s' \rightarrow^* s\}$ *of predecessors of states in I. A finite basis of* $Pred^*(I)$ *is computable.*

In the remainder of the section, we assume a given universe U of component types; so we can consider that the sets of possible component types $\mathscr{T}$ and of possible internal states q are both finite. We will resort to the theory of WSTS by considering an abstract model of configurations in which bindings are not taken into account.

Definition 10 (Abstract Configuration). *An abstract configuration $\mathscr{B}$ is a finite multiset of pairs $\langle \mathscr{T}, q \rangle$ where $\mathscr{T}$ is a component type and q is a corresponding state. We use $Conf$ to denote the set of abstract configurations.*

A concretization of an abstract configuration is simply a correct configuration that for every component-type and state pair $\langle \mathscr{T}, q \rangle$ has as many instances of component $\mathscr{T}$ in state q as pairs $\langle \mathscr{T}, q \rangle$ in the abstract configuration.

Definition 11 (Concretization). *Given an abstract configuration $\mathscr{B}$ we say that a correct configuration $\mathscr{C} = \langle U, Z, S, B \rangle$ is one concretization of $\mathscr{B}$ if there exists a bijection f from the multiset $\mathscr{B}$ to Z s.t. $\forall \langle \mathscr{T}, q \rangle \in \mathscr{B}$ we have that $S(f(\langle \mathscr{T}, q \rangle)) = \langle \mathscr{T}, q \rangle$. We denote with $\gamma(\mathscr{B})$ the set of concretizations of $\mathscr{B}$. We say that an abstract configuration $\mathscr{B}$ is correct if it has at least one concretization (formally $\gamma(\mathscr{B}) \neq \emptyset$).*

An interesting property of an abstract configuration is that from one of its concretizations it is possible to reach via bind and unbind actions all the other concretizations up to instance renaming. This is because it is always possible to switch one binding from one provide port to another one by adding a binding to the new port and then removing the old binding.

Property 1. Given an abstract configuration $\mathscr{B}$ and configurations $\mathscr{C}_1, \mathscr{C}_2 \in \gamma(\mathscr{B})$ there exists $\alpha_1, \ldots, \alpha_n$ sequence of binding and unbinding actions s.t. $\mathscr{C}_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \mathscr{C} \equiv \mathscr{C}_2$.

We now move to the definition of our quasi-ordering on abstract configurations. In order to be compatible with the notion of correctness we cannot adopt the usual multiset inclusion ordering. In fact, the addition of one component to a correct configuration could introduce a conflict. If the type-state pair of the added component was absent in the configuration, the conflict might be with a component of a different type-state. If the type-state pair was present in a single copy, the conflict might be with that component if the considered type-state pair activates one provide and one conflict port on the same interface. This sort of self-conflict is revealed when there are at least two instances, as one component cannot be in conflict with itself. If the type-state pair was already present in at least two copies, no new conflicts can be added otherwise such conflicts were already present in the configuration (thus contradicting its correctness).

In the light of the above observation, we define an ordering on configurations that corresponds to the product of three orderings: the identity on the set of type-state pairs that are absent, the identity on the pairs that occurs in one instance, and the multiset inclusion for the projections on the remaining type-state pairs.

Definition 12 ($\leq$). *Given a pair $\langle \mathscr{T}, q \rangle$ and an abstract configuration $\mathscr{B}$, let $\#_{\mathscr{B}}(\langle \mathscr{T}, q \rangle)$ be the number of occurrences in $\mathscr{B}$ of the pair $\langle \mathscr{T}, q \rangle$. Given two abstract configurations $\mathscr{B}_1, \mathscr{B}_2$ we write $\mathscr{B}_1 \leq \mathscr{B}_2$ if for every component type $\mathscr{T}$ and state q we have that $\#_{\mathscr{B}_1}(\langle \mathscr{T}, q \rangle) = \#_{\mathscr{B}_2}(\langle \mathscr{T}, q \rangle)$ when $\#_{\mathscr{B}_1}(\langle \mathscr{T}, q \rangle) \in \{0, 1\}$ or $\#_{\mathscr{B}_2}(\langle \mathscr{T}, q \rangle) \in \{0, 1\}$, and $\#_{\mathscr{B}_1}(\langle \mathscr{T}, q \rangle) \leq \#_{\mathscr{B}_2}(\langle \mathscr{T}, q \rangle)$ otherwise.*

As discussed above, this ordering is compatible with correctness.

Property 2. If an abstract configuration $\mathscr{B}$ is correct than all the configurations $\mathscr{B}'$ such that $\mathscr{B} \leq \mathscr{B}'$ are also correct.

Another interesting property of the $\leq$ quasi-ordering is that from one concretization of an abstract configuration, it is always possible to reconfigure it to reach a concretization of a smaller abstract configuration. In this case it is possible to first add from the starting configuration the bindings that are present in the final configuration. Then the extra components present in the starting configuration can be deleted because not needed to guarantee correctness (they are instances of components that remain available in at least two copies). Finally the remaining extra bindings can be removed.

Property 3. Given two abstract configurations $\mathscr{B}_1, \mathscr{B}_2$ s.t. $\mathscr{B}_1 \leq \mathscr{B}_2$, $\mathscr{C}_1 \in \gamma(\mathscr{B}_1)$, and $\mathscr{C}_2 \in \gamma(\mathscr{B}_2)$ we have that there exists a reconfiguration run $\mathscr{C}_2 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \mathscr{C} \equiv \mathscr{C}_1$.

We have that $\leq$ is a wqo on *Conf* because, as we consider finitely many component type-state pairs, the three distinct orderings that compose $\leq$ are themselves wqo.

Lemma 1. *$\leq$ is a wqo over Conf.*

We now define a transition system on abstract reconfigurations and prove it is a WSTS with respect to the ordering defined above.

Definition 13 (Abstract reconfigurations). *We write $\mathscr{B} \to \mathscr{B}'$ if there exists $\mathscr{C} \xrightarrow{\alpha} \mathscr{C}'$ for some $\mathscr{C} \in \gamma(\mathscr{B})$ and $\mathscr{C}' \in \gamma(\mathscr{B}')$.*

By Property 3 and Lemma 1 we have the following.

Lemma 2. *The transition system $(Conf, \to, \leq)$ is a WSTS.*

The following lemma is rather technical and it will be used to prove that $(Conf, \to, \leq)$ has effective pred-basis. Intuitively it will allow us to consider, in the computation of the predecessors, only finitely many different state change actions.

Lemma 3. *Let k be the number of distinct component type-state pairs. If $\mathscr{B}_1 \to \mathscr{B}_2$ then there exists $\mathscr{B}'_1 \to \mathscr{B}'_2$ such that $\mathscr{B}'_1 \leq \mathscr{B}_1$, $\mathscr{B}'_2 \leq \mathscr{B}_2$ and $|\mathscr{B}'_2| \leq 3k + 2k^2$.*

Proof. If $|\mathscr{B}_2| \leq 3k + 2k^2$ the thesis trivially holds. Consider now $|\mathscr{B}_2| > 3k + 2k^2$ and a transition $\mathscr{C}_1 \xrightarrow{\alpha} \mathscr{C}_2$ such that $\mathscr{C}_1 \in \gamma(\mathscr{B}_1)$ and $\mathscr{C}_2 \in \gamma(\mathscr{B}_2)$. Since $|\mathscr{B}_2| > 3k$ there are three components z_1, z_2 and z_3 having the same component type and internal state. We consider two subcases.
Case 1. z_1, z_2 and z_3 do not perform a state change in the action α. W.l.o.g we can assume that z_3 does not appear in α (this is not restrictive because at most two components that do not perform a state change can occur in an action). We can now consider the configuration $\mathscr{C}'_1$ obtained by $\mathscr{C}_1$ after removing z_3 (if there are bindings connected to provide ports of z_3, these can be rebound to ports of z_1 or z_2). Consider now $\mathscr{C}'_1 \xrightarrow{\alpha} \mathscr{C}'_2$ and the corresponding abstract configurations $\mathscr{B}'_1$ and $\mathscr{B}'_2$. It is easy to see that $\mathscr{B}'_1 \to \mathscr{B}'_2$,

$\mathscr{B}'_1 \leq \mathscr{B}_1$, $\mathscr{B}'_2 \leq \mathscr{B}_2$ and $|\mathscr{B}'_2| < |\mathscr{B}_2|$. If $|\mathscr{B}'_2| \leq 3k+2k^2$ the thesis is proved, otherwise we repeat this deletion of components.
Case 2. There are no three components of the same type-state that do not perform a state change. Since $|\mathscr{B}_2| > 2k^2+2$ we have that α is a state change involving strictly more than $2k^2$ components. This ensures the existence of three components z'_1, z'_2 and z'_3 of the same type that perform the same state change from q to q'. As in the previous case we consider the configuration $\mathscr{C}'_1$ obtained by $\mathscr{C}_1$ after removing z'_3 and α' the state change similar to α but without the state change of z'_3. Consider now $\mathscr{C}'_1 \xrightarrow{\alpha'} \mathscr{C}'_2$ and the corresponding abstract configurations $\mathscr{B}'_1$ and $\mathscr{B}'_2$. As above, $\mathscr{B}'_1 \leq \mathscr{B}_1$, $\mathscr{B}'_2 \leq \mathscr{B}_2$ and $|\mathscr{B}'_2| < |\mathscr{B}_2|$. If $|\mathscr{B}'_2| \leq 3k+2k^2$ the thesis is proved, otherwise we repeat the deletion of components. □

We are now in place to prove that $(Conf, \rightarrow, \leq)$ has effective pred-basis.

Lemma 4. *The transition system* $(Conf, \rightarrow, \leq)$ *has effective pred-basis.*

Proof. We first observe that given an abstract configuration the set of its concretizations up to configuration equivalence is finite, and that given a configuration $\mathscr{C}$ the set of preceding configurations $\mathscr{C}'$ such that $\mathscr{C}' \xrightarrow{\alpha} \mathscr{C}$ is also finite (and effectively computable). Consider now an abstract configuration $\mathscr{B}$. We now show how to compute a finite basis for $\uparrow Pred(\uparrow \mathscr{B})$. First of all we consider the configuration $\mathscr{B}$ if $|\mathscr{B}| > 3k+2k^2$, the (finite) set of configurations $\mathscr{B}'$ such that $\mathscr{B} \leq \mathscr{B}'$ and $|\mathscr{B}'| \leq 3k+2k^2$ otherwise. Then we consider the (finite) set of concretizations of all such abstract configurations. And finally we compute the (finite) set of the preceding configurations of all such concretizations. The set of abstract configuration corresponding to the latter is a finite basis for $\uparrow Pred(\uparrow \mathscr{B})$ as a consequence of Lemma 3. □

We are finally ready to prove our decidability result.

Theorem 1. *The reconfigurability problem in* Aeolus core *is decidable.*

Proof. Let k be the number of distinct component type-state pairs according to the considered universe of component types. We first observe that if there exists a correct configuration containing a component of type $\mathscr{T}$ in state q then it is possible to obtain via some binding, unbinding, and delete actions another correct configuration with k or less components. Hence, given a component type $\mathscr{T}$ and a state q, the number of target configurations that need to be considered is finite. Moreover, given a configuration $\mathscr{C}' \in \gamma(\mathscr{B}')$ there exists a reconfiguration run from $\mathscr{C} \in \gamma(\mathscr{B})$ to $\mathscr{C}'$ iff $\mathscr{B} \in Pred^*(\uparrow \mathscr{B}')$.

To solve the reconfigurability problem it is therefore possible to consider only the (finite set of) abstractions of the target configurations. For each of them, say $\mathscr{B}'$, by Proposition 1, Lemma 2 and Lemma 4 we know that a finite basis for $Pred^*(\uparrow \mathscr{B}')$ can be computed. It is sufficient to check whether at least one of the abstract configurations in such basis is $\leq$ w.r.t. the abstraction of the initial configuration. □

4 Reconfigurability is ExpSpace-hard in Aeolus core

We prove that the reconfigurability problem in Aeolus core is ExpSpace-hard by reduction from the coverability problem in Petri nets, a problem which is indeed known to be ExpSpace-complete [11,13]. We start with some background on Petri nets.

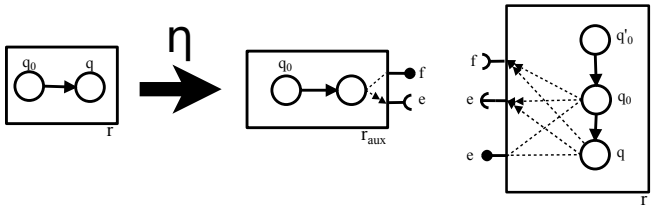

Fig. 2. Example of a component type transformation $\eta(\,)$

A *Petri net* is a tuple $N = (P,T,m_0)$, where P and T are finite sets of *places* and *transitions*, respectively. A finite multiset over the set P of places is called a *marking*, and m_0 is the initial marking. Given a marking m and a place p, we say that the place p contains a number of *tokens* equal to the number of instances of p in m. A transition $t \in T$ is a pair of markings denoted with ${}^\bullet t$ and $t^\bullet$. A transition t can fire in the marking m if ${}^\bullet t \subseteq m$ (where $\subseteq$ is multiset inclusion); upon transition firing the new marking of the net becomes $n = (m \setminus m') \uplus m''$ (where $\setminus$ and $\uplus$ are the difference and union operators for multisets, respectively). This is written as $m \Rightarrow n$. We use $\Rightarrow^*$ to denote the reflexive and transitive closure of $\Rightarrow$. We say that m' is *reachable from* m if $m \Rightarrow^* m'$. The *coverability* problem for marking m consists of checking whether $m_0 \Rightarrow^* m'$ for some $m \subseteq m'$.

We now discuss how to encode Petri nets in Aeolus core component types. Before entering into the details we observe that given a component type $\mathscr{T}$ it is always possible to modify it in such a way that its instances are persistent and unique. The uniqueness constraint can be enforced by allowing all the states of the component type to provide a new port with which they are in conflict. To avoid the component deletion it is sufficient to impose its reciprocal dependence with a new type of component. When this dependence is established the components be deleted without violating it. In Fig. 2 we show an example of how a component type having two states can be modified in order to reach our goal. A new auxiliary initial state q'_0 is created. The new port e ensures that the instances of type $\mathscr{T}$ in a state different from q'_0 are unique. The require port f provided by a new component type $\mathscr{T}_{aux}$ forbids the deletion of the instances of type $\mathscr{T}$, if they are not in state q'_0. We assume that the ports e and f are fresh. We can therefore consider w.l.o.g. components that, when deployed, are unique and persistent. Given a component type $\mathscr{T}$ we denote this component type transformation with $\eta(\mathscr{T})$.

We now describe how to encode a Petri net in the Aeolus core model. We will use three types of components: one modeling the tokens, one for transitions and one for defining a counter. The components for transitions and the counter are unique and persistent, while those for the tokens cannot be unique because the number of tokens in a Petri net can be unbounded. The simplest component is the one used to model a token in a given place. Intuitively one token in a place is encoded as one instance of a corresponding component type in an *on* state. There could be more than one of these components deployed simultaneously representing multiple tokens in a place. In Fig. 3a we represent the component type for the tokens in the place p of the Petri net. The initial state is the *off* state. The token could be created following a protocol consisting of requiring the port a_p and then providing the port b_p to signal the change of status. Similarly a token can be deleted requiring the port c_p and then providing the port d_p.

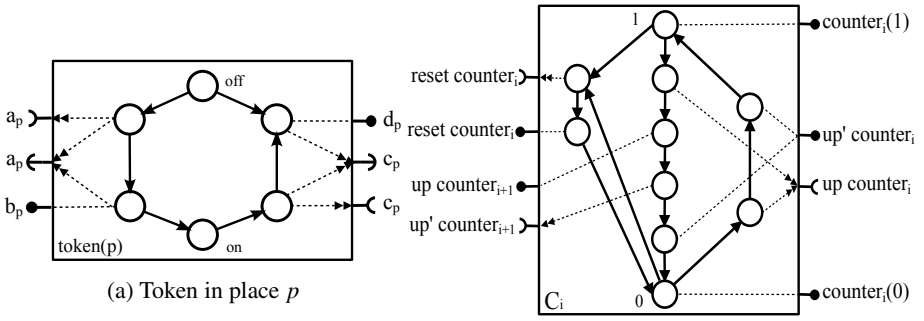

Fig. 3. Token and counter component types

Even if multiple instances of the token component can be deployed simultaneously, the conflict ports a_p and c_p guarantee that only one at a time can initiate the protocol to change its state. We denote with *token(p)* the component type representing the tokens in the place p.

In order to model the transitions with component types without having an exponential blow up of the size of the encoding we need a mechanism to count up to a fixed number. Indeed a transition can consume and produce up to a given number of tokens. To count a number up to n we will use $C_1, \ldots, C_{\lceil \log(n) \rceil}$ components; every C_i will represent the i-th less significant bit of the binary representation of the counter that, for our purposes, needs just to support the increment and reset operations. In Fig. 3b we represent one of the bits implementing the counter. The initial state is 0. To reset the bit it is possible to provide the *reset counter$_i$* port while to increment it the *up counter$_i$* should be provided. If the bit is in state 1 the increment will trigger the increment of the next bit except for the component representing the most significant bit that will never need to do that. We transform all the component types representing the counter using the η transformation to ensure uniqueness and persistence of its instances. The instance of $\eta(C_i)$ can be used to count how many tokens are consumed or produced checking if the right number is reached via the ports *counter$_i$(1)* and *counter$_i$(0)*.

A transition can be represented with a single component interacting with token and counter components. The state changes of the transition component can be intuitively divided in phases. In each of those phases a fixed number of tokens from a given place is consumed or produced. The counter is first reset providing the *reset counter$_i$* and requiring the *reset$'$ counter$_i$* ports for all the counter bits. Then a cycle starts incrementing the counter providing and requiring the ports *up counter*$_1$ and *up$'$ counter*$_1$ and consuming or producing a token. The production of a token in place p is obtained providing and requiring ports a_p and b_p while the consumption providing and requiring the ports c_p and d_p. The phase ends when all the bits of the counter represent in binary the right number of tokens that need to be consumed or produced. If instead at least one bit is wrong the cycle restarts. In Fig. 4 we depict the phase of a consumption of n tokens.

Starting from the initial state of the component representing the transition, the consumption phases need to be performed first. When the final token has been produced

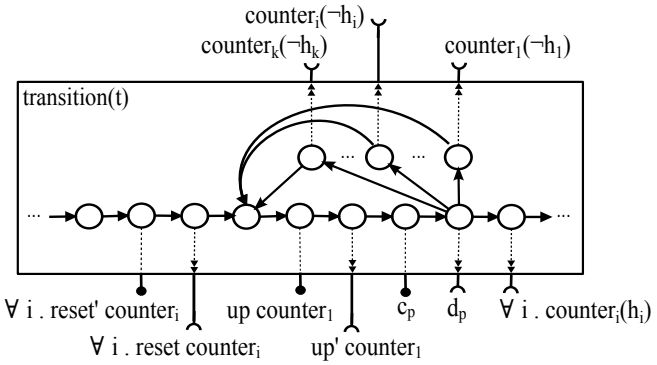

Fig. 4. Consumption phase of n tokens from place p for a transition t ($k = \lceil \log(n) \rceil$ and h_i is the i-th least significative bit of the binary representation of n)

the transition component can restart from the initial state. Given a transition t we will denote with *transition(t)* the component type explained above.

Definition 14 (Petri net encoding in Aeolus core**).** *Given a Petri net $N = (P,T,m_0)$ if n is the largest number of tokens that can be consumed or produced by a transition in T, the encoding of N in* Aeolus core *is the set of component types $\Gamma_N = \{token(p) \mid p \in P\} \cup \{\eta(C_i) \mid i \in [1..\lceil \log(n) \rceil]\} \cup \{\eta(transition(t)) \mid t \in T\}$.*

An important property of the previous encoding is that it is polynomial w.r.t. the size of the Petri net. This is due to the fact that the counter and place components have a constant amount of states and ports while the transition components have a number of states that grows linearly w.r.t. the number of places involved in a transition.

The proof that the reconfiguration problem for Aeolus core is ExpSpace-hard thus follows from the following correspondence between a Petri net N and its set of component types Γ_N: every computation in N can be faithfully reproduced by a corresponding reconfiguration run on the components types Γ_N; every reconfiguration run on Γ_N corresponds to a computation in N excluding the possibility for components of kind $token(p)$ to be deleted (because η is not applied to those components) and of components $transition(t)$ to execute only partially the consumption of the tokens (because e.g. some token needed by the transition is absent). In both cases, the effect is to reach a configuration in which some of the token was lost during the reconfiguration run, but this is not problematic as we deal with coverability. In fact, if a configuration is reached with at least some tokens, then also the corresponding Petri nets will be able to reach a marking with at least those tokens (possibly more).

Theorem 2. *The reconfiguration problem for* Aeolus core *is ExpSpace-hard.*

5 Related Work and Conclusions

Engage [8] is very close to Aeolus purposes: it provides a declarative language to define resource configurations and a deployment engine. However, it lacks conflicts.

This might make a huge computational differences, as it is precisely the introduction of conflicts that makes reconfigurability ExpSpace-hard in Aeolus core (the problem is polynomial in Aeolus$^-$ [6]). ConfSolve [9] is a DSL used to specify system configurations with constraints suitable for modern Constraint Satisfaction Problems solvers. ConfSolve allocates virtual machines to physical ones considering constraints like CPU, RAM, ... This differs from reconfigurability in Aeolus. Package-based software management [1,5] is a degenerate case of Aeolus reconfigurability. Package managers are used to compute a new configuration, but they use simple heuristics to reach it, ignoring transitive incoherences met during deployment.

In this work we have studied the impact of adding conflicts to a realistic component model, onto the complexity of reconfigurability: the problem remains decidable—while in other models, like Petri nets, the addition of tests-for-absence makes the model Turing powerful—but becomes ExpSpace-hard.

We consider our decidability and hardness proofs useful for at least two future intertwined research directions. On the one hand, we plan to extend existing tools with techniques inspired by our decidability proof in order to also deal with conflicts and produce a reconfiguration run. On the other hand, the hardness proof sheds some light on the specific combination of component model features that make the reconfigurability problem ExpSpace-hard. We plan to investigate realistic restrictions on the Aeolus component model for which efficient reconfigurability algorithms could be devised.

References

1. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Dependency solving: a separate concern in component evolution management. J. Syst. Software 85, 2228–2240 (2012)
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321. IEEE (1996)
3. Clayberg, E., Rubel, D.: Eclipse Plug-ins, 3rd edn. Addison-Wesley (2008)
4. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Component reconfiguration in the presence of conflicts. Tech. rep. Aeolus Project (2013), `http://hal.archives-ouvertes.fr/hal-00816468`
5. Di Cosmo, R., Trezentos, P., Zacchiroli, S.: Package upgrades in FOSS distributions: Details and challenges. In: HotSWup 2008 (2008)
6. Di Cosmo, R., Zacchiroli, S., Zavattaro, G.: Towards a formal component model for the cloud. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 156–171. Springer, Heidelberg (2012)
7. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256, 63–92 (2001)
8. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: a deployment management system. In: PLDI 2012: Programming Language Design and Implementation, pp. 263–274. ACM (2012)
9. Hewson, J.A., Anderson, P., Gordon, A.D.: A declarative approach to automated configuration. In: LISA 2012: Large Installation System Administration Conference, pp. 51–66 (2012)
10. Kanies, L.: Puppet: Next-generation configuration management. The USENIX Magazine 31(1), 19–25 (2006)
11. Lipton, R.J.: The Reachability Problem Requires Exponential Space. Research report 62, Department of Computer Science, Yale University (1976)
12. OSGi Alliance: OSGi Service Platform, Release 3. IOS Press, Inc. (2003)
13. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoret. Comp. Sci. 6, 223–231 (1978)